Session 5

White-Box Testing (3)

xfzhang@suda.edu.cn

Objectives

- Review of Session 3-4
  - Basic Concepts of White-Box Testing
  - Logic Coverage
  - Control Flow Graph
  - Basis Path Testing
  - Loop Testing
  - Data Flow Testing

3.1 Basic Concepts

- White-box testing must follow several principles:
  - All independent path in a module must be implemented at least once. (Basis path testing)
  - All logic values require two test cases: true and false. (Logic coverage)
  - Inspection procedures of internal data structure, and ensuring the effectiveness of its structure.

    (Static Testing + Data Flow Testing)
  - Run all cycles within operational range. (Loop testing)

3.2 Logic Coverage

- Logic coverage
  - Statement coverage
  - Decision coverage
  - Condition coverage
  - Condition/decision coverage
  - Condition combination coverage
  - Path coverage
  - Complete coverage

3.3 Control Flow Graph

- Concept
  - During procedure design , in order to more prominent the control flow structure, the procedure can be simplified, the simplified graph is called control flow graph. (vs. program flow graph)
  - On a flow graph:
    - Arrows called *edges* represent flow of control.
    - Circles called *nodes* represent one or more actions.
    - Areas bounded by edges and nodes called *regions*.
    - A *predicate node* is a node containing a condition.

3.4 Basis Path Testing

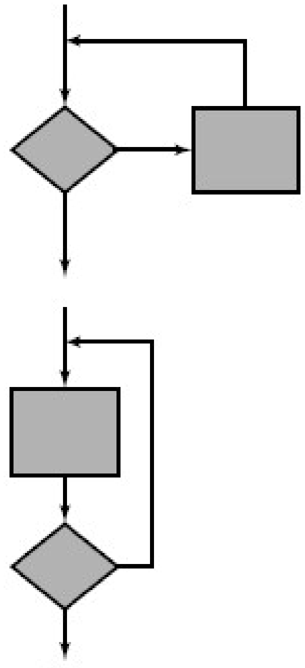
- How to design test cases for basis path testing
  - Using the design or code, draw the corresponding control flow graph.
  - Determine the cyclomatic complexity of the flow graph.
  - Determine a basis set of independent paths.
  - Prepare test cases that will force execution of each path in the basis set.

3.5 Loop Testing

- Simple Loops
- Nested Loops
- Concatenated Loops
- Unstructured Loops

3.5 Loop Testing – Simple Loops

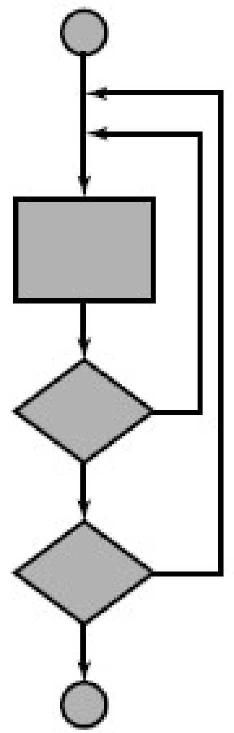
- The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.
  - Skip the loop entirely.
  - Only one pass through the loop.
  - Two passes through the loop.
  - m passes through the loop where m < n.
  - n - 1, n, n + 1 passes through the loop



Simple loops

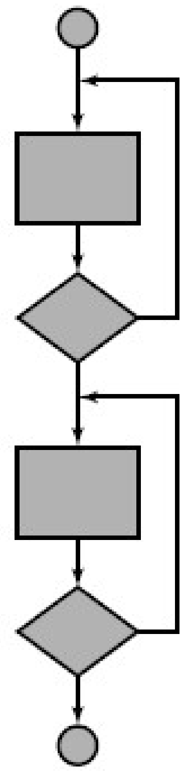
3.5 Loop Testing – Nested Loops

- Start at the innermost loop. Set all other loops to minimum values.
- Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values.
- Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
- Continue until all loops have been tested.


Nested loops

3.5 Loop Testing – Concatenated Loops

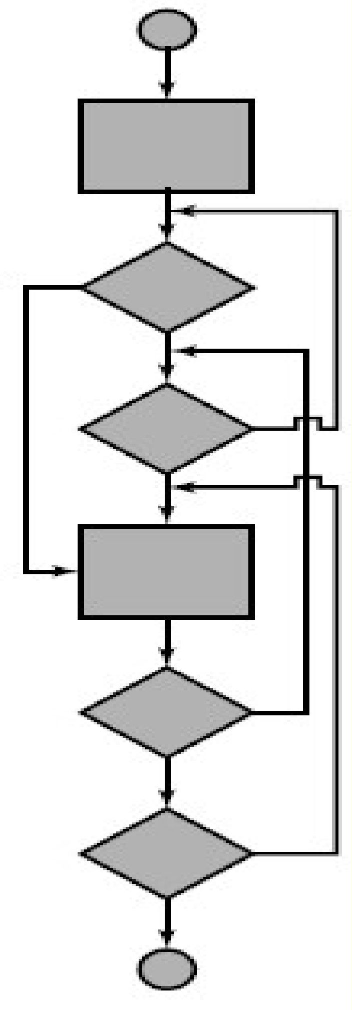
- If each of the loops is independent of the other:
  - Concatenated loops can be tested using the approach defined for simple loops,
- Else
  - the approach applied to nested loops is recommended.



Concatenated loops

3.5 Loop Testing – Unstructured Loops

- Suggestion: whenever possible, this kind of loops should be redesigned.
- avoid to use “go to” control construct.



Unstructured loops

3.5 Loop Testing

* The easy way to test loops is using the method Z path coverage.
* Z path coverage method is a program in Loop structure will be simplified into IF structure test method.
* Simplify cycle test only once, or zero times. In this case, the effect of loop structure and IF branches are same, namely the loop body or execution, or skip.

3.5 Loop Testing

- Loop Testing & Basis Path Testing

(3) for(int i=1; i<arraty_size; i++)

3.1 3.2 3.3

3.5 Loop Testing

Insertion Sort

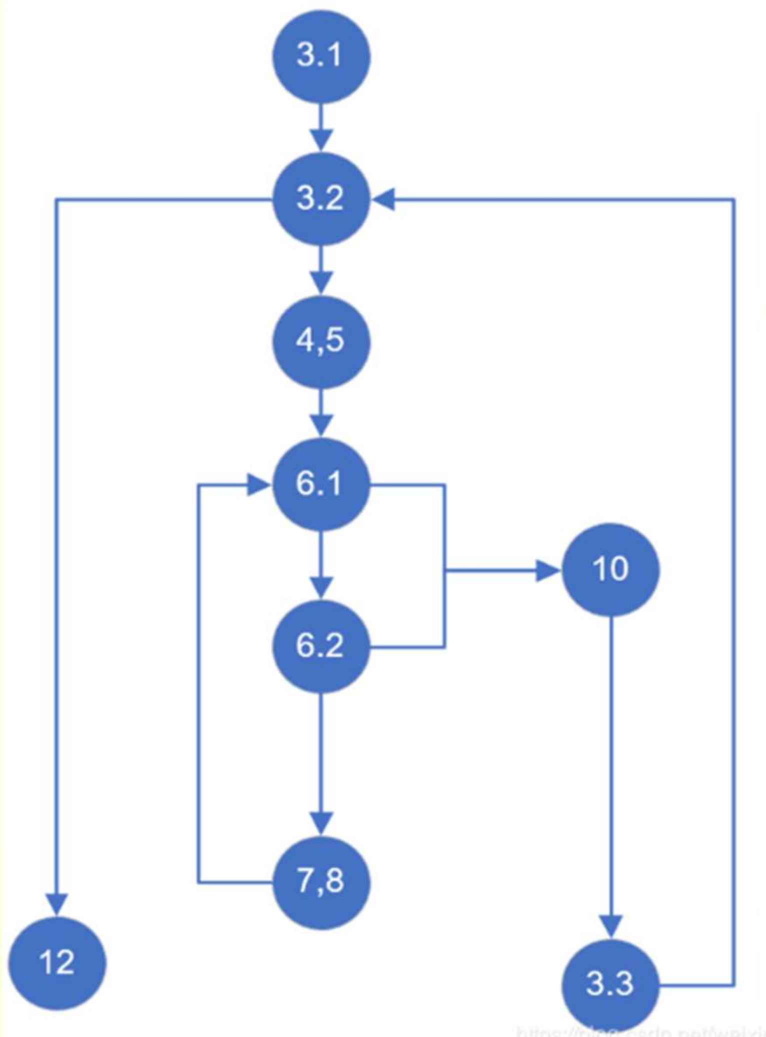
```
(1)  void insertionSort(int numbers[], int array_size)( {
(2)      int i, j, index;
                 3.1      3.2      3.3
(3)  for (i=1; i < array_size; i++)      {
(4)      index = numbers[i];
(5)      j = i;
                 6.1                  6.2
(6)      while ((j > 0) && (numbers[j-1] > index))  {
(7)      numbers[j] = numbers[j-1];
(8)      j = j - 1;
(9)      }
(10)     numbers[j] = index;
(11)  }
(12) }
```


Update CFG

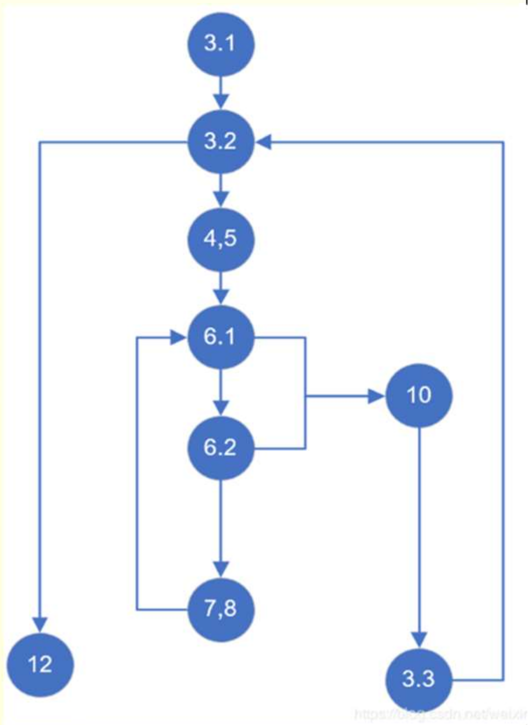
3.5 Loop Testing

Insertion Sort

```
(1)  void insertionSort(int numbers[], int array_size)( {
(2)     int i, j, index;
                    3.1      3.2      3.3
(3)  for (i=1; i < array_size; i++)      {
(4)     index = numbers[i];
(5)     j = i;
                 6.1                  6.2
(6)     while ((j > 0) && (numbers[j-1] > index))  {
(7)     numbers[j] = numbers[j-1];
(8)     j = j - 1;
(9)     }
(10)    numbers[j] = index;
(11)  }
(12) }
```



3.5 Loop Testing



Insertion Sort

```
(1)  void insertionSort(int numbers[], int array_size)( {
(2)      int i, j, index;
              3.1          3.2          3.3
(3)  for (i=1; i < array_size; i++)      {
(4)      index = numbers[i];
(5)      j = i;
                 6.1                 6.2
(6)      while ((j > 0) && (numbers[j-1] > index))  {
(7)      numbers[j] = numbers[j-1];
(8)      j = j - 1;
(9)      }
(10)     numbers[j] = index;
(11)  }
(12) }
```

路径1：3.1–3.2–12
路径2：3.1–3.2–4,5–6.1–10–3.3–3.2–12
路径3：3.1–3.2–4,5–6.1–6.2–10–3.3–3.2–12
路径4：3.1–3.2–4,5–6.1–6.2–7,8–6.1–10–3.3–3.2–12

程序中6.1在未进入while循环自减时j=1恒成立，故不存在任何数据能使程序执行路径2.

3.6 Data Flow Testing

- The data flow testing method selects test paths of a program according to the locations of definitions, uses and deletions of variables in the program
- Data flow testing is a powerful tool to detect improper use of data values due to coding errors
  - Incorrect assignment or input statement
  - Definition is missing (use of null definition)
  - Predicate is faulty (incorrect path is taken which leads to incorrect definition)

Variable Definitions and Uses

- A program variable is **DEFINED** when it appears:
  - on the *left* hand side of an assignment statement (e.g., Y := 17)
  - in an input statement (e.g., input(Y))
  - as an OUT parameter in a subroutine call (e.g., DOIT(X:IN,Y:OUT))

  - 将数据存储起来, 存储单元的内容改变

Variable Definitions and Uses

- A program variable is **USED** when it appears:
  - on the *right* hand side of an assignment statement (e.g., Y := X+17)
  - as an IN parameter in a subroutine or function call (e.g., Y := SQRT(X))
  - in the predicate of a branch statement (e.g., if X>0 then...)

  - 将数据取出来，存储单元的内容不变

Variable Definitions and Uses

- Use of a variable in the predicate of a branch statement is called a *predicate-use* **("p-use").** Any other use is called a *computation-use* **("c-use").**
- For example, in the program statement:

```
If (X>0) then
    print(Y)
end_if_then
```

there is a p-use of X and a c-use of Y.

Variable Definitions and Uses

- A variable can also be used and then re-defined in a single statement when it appears:
  - on *both* sides of an assignment statement (e.g., Y := Y+X)
  - as an IN/OUT parameter in a subroutine call (e.g., INCREMENT(Y:IN/OUT))

3.6 Data Flow Testing

- Variables that contain data values have a defined life cycle: defined, used, killed (destroyed).
- The "scope" of the variable

```
{            // begin outer block
   int x;    // x is defined as an integer within this outer block
  ...;       // x can be accessed here
    {        // begin inner block
    int y;   // y is defined within this inner block
  ...;       // both x and y can be accessed here
    }        // y is automatically destroyed at the end of this block
    ...;     // x can still be accessed, but y is gone
 }   // x is automatically destroyed
```

3.6 Data Flow Testing

- Three possibilities exist for the first occurrence of a variable through a program path:
  - ~d - the variable does not exist (indicated by the ~), then it is defined (d)
  - ~u - the variable does not exist, then it is used (u): c-use / p-use
  - ~k - the variable does not exist, then it is killed or destroyed (k)

3.6 Data Flow Testing

- List 9 pairs of defined (d), used (u), and killed (k):
  - dd - Defined and defined again—not invalid but suspicious. Probably a programming error.
  - du - Defined and used—perfectly correct. The normal case.
  - dk - Defined and then killed—not invalid but probably a programming error.
  - ud - Used and defined—acceptable.
  - uu - Used and used again—acceptable.
  - uk - Used and killed—acceptable.
  - kd - Killed and defined—acceptable. A variable is killed and then redefined.
  - ku - Killed and used—a serious defect. Using a variable that does not exist or is undefined is always an error.
  - kk - Killed and killed—probably a programming error

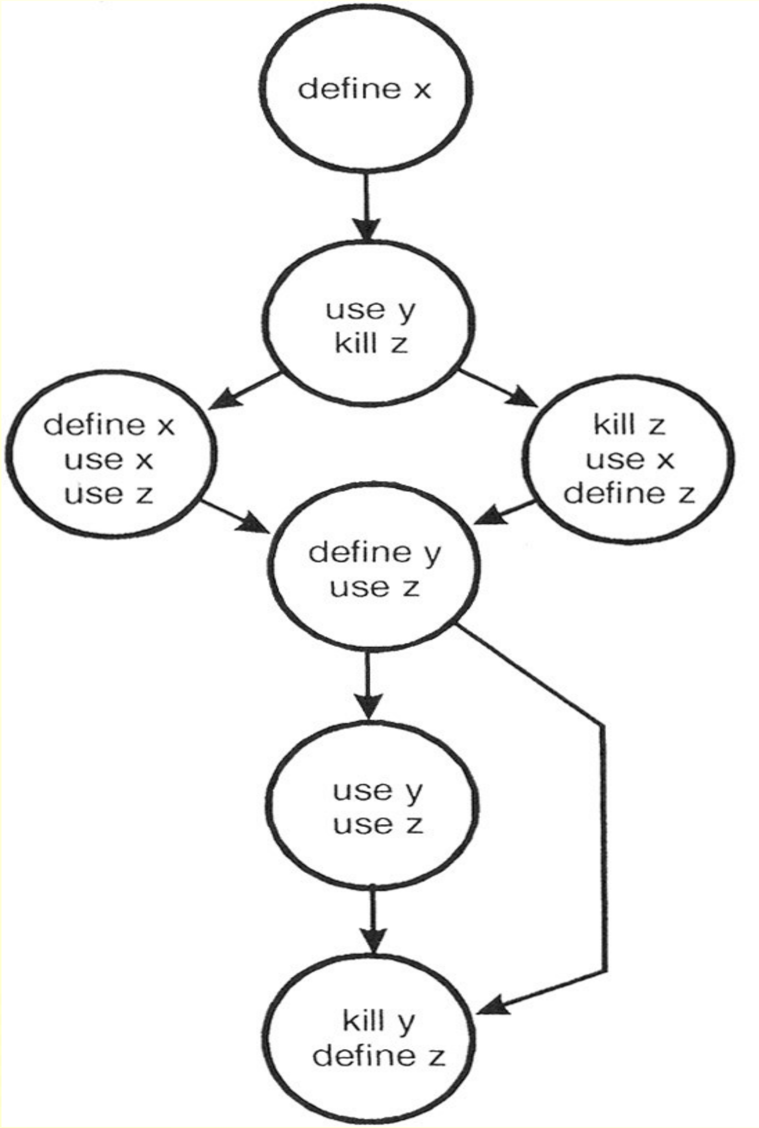
3.6 Data Flow Testing

* A data flow graph is similar to a control flow graph in that it shows the processing flow through a module.

  In addition, it details the definition, use, and destruction of each of the module's variables.
* Technique
  * Construct diagrams
  * Perform a static test of the diagram
  * Perform dynamic tests on the module

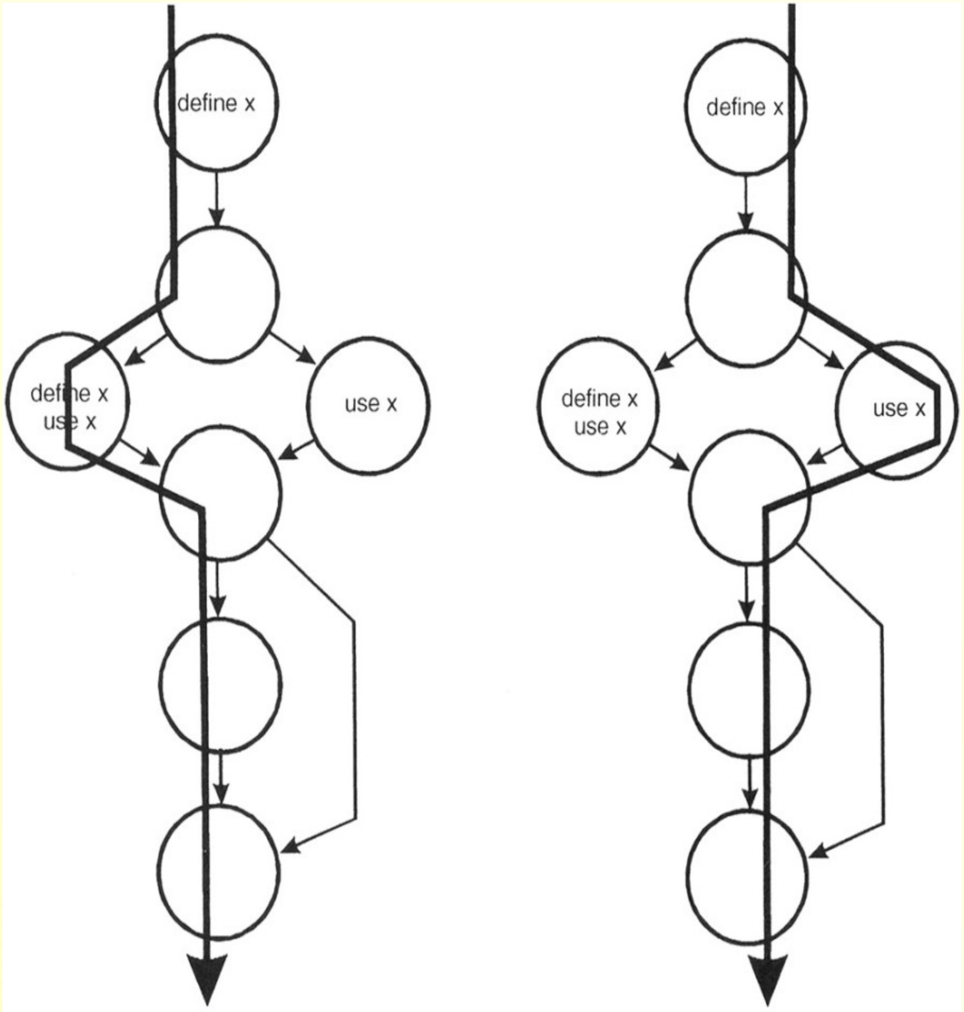
3.6 Data Flow Testing

- For each variable within the module we will examine define-use-kill patterns along the control flow paths.



3.6 Data Flow Testing

- The define-use-kill patterns for x (taken in pairs as we follow the paths) are:
  - ~define - correct, the normal case
  - define-define - suspicious, perhaps a programming error
  - define-use - correct, the normal case

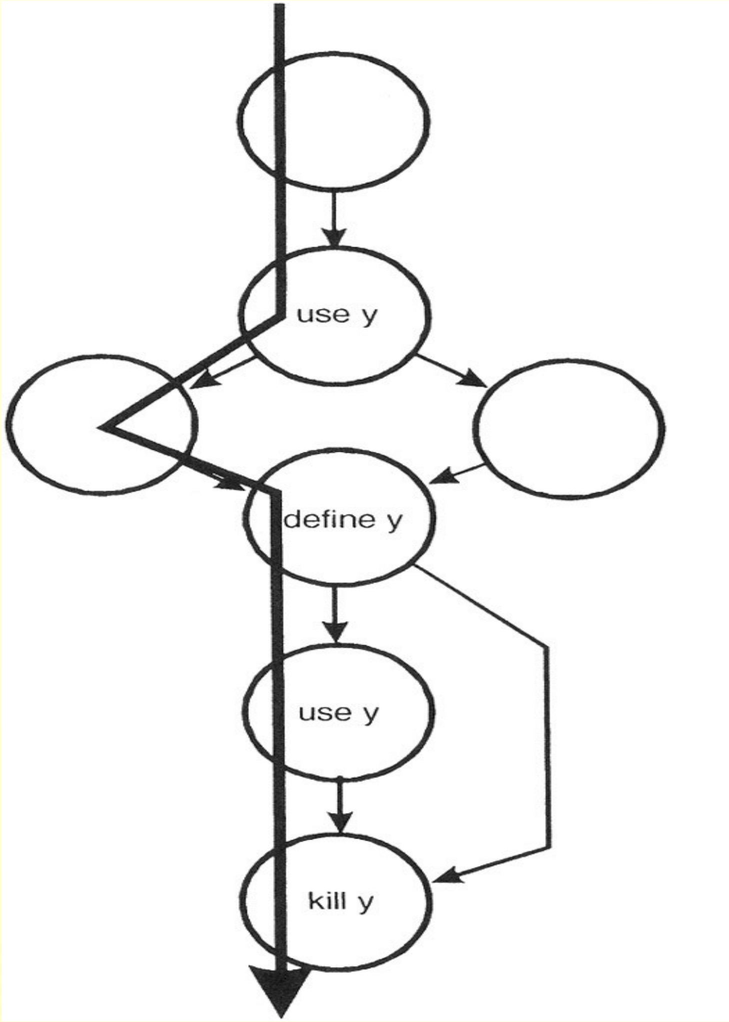


3.6 Data Flow Testing

* Exercise:
  * List the define-use-kill patterns for y and z (taken in pairs as we follow the paths)

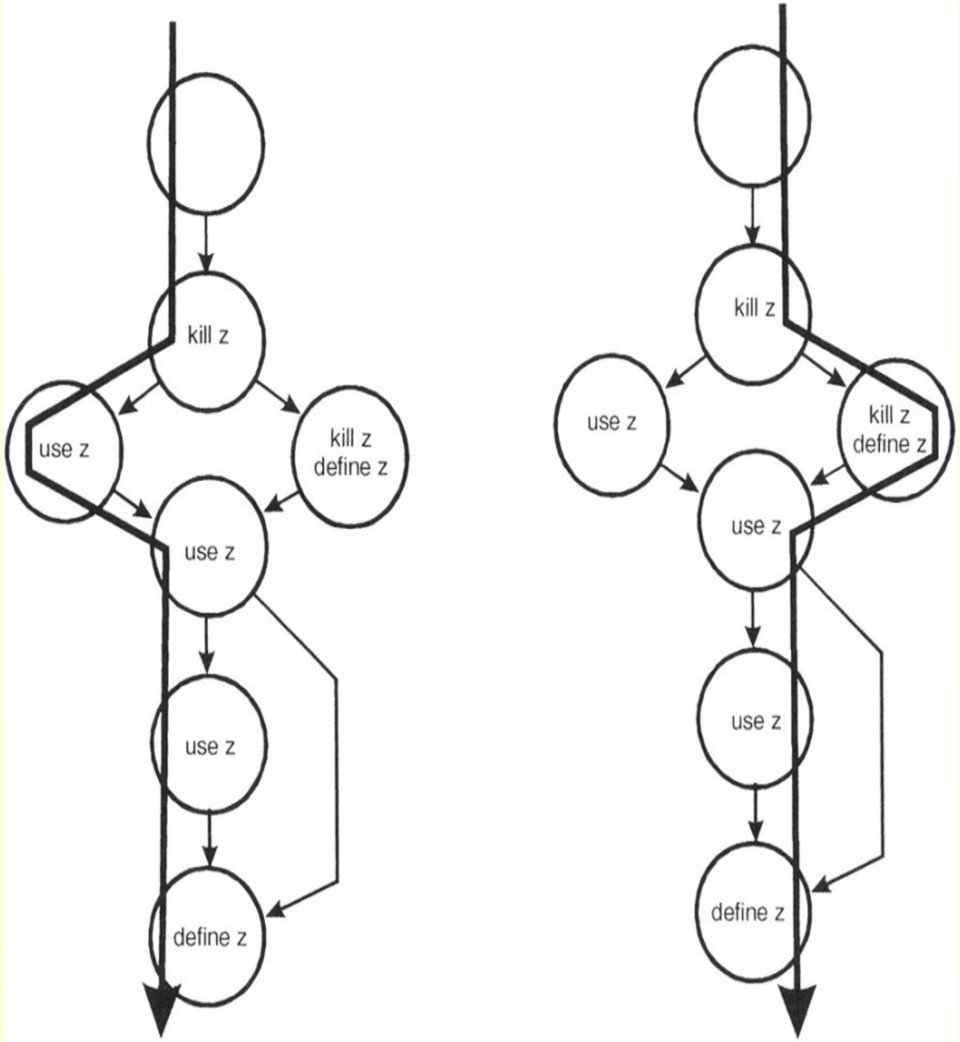
3.6 Data Flow Testing

- The define-use-kill patterns for y (taken in pairs as we follow the paths) are:
  - ~use - major blunder
  - use-define - acceptable
  - define-use - correct, the normal case
  - use-kill - acceptable
  - define-kill - probable programming error



3.6 Data Flow Testing

* The define-use-kill patterns for z (taken in pairs as we follow the paths) are:
  * ~kill - programming error
  * kill-use - major blunder
  * use-use - correct, the normal case
  * use-define - acceptable
  * kill-kill - probably a programming error
  * kill-define - acceptable
  * define-use - correct, the normal case



3.6 Data Flow Testing

- In performing a static analysis on this data flow model the following problems have been discovered:
  - x: define-define
  - y: ~use
  - y: define-kill
  - z: ~kill
  - z: kill-use
  - z: kill-kill

- In session 3-5, you have learned
  - Basic Concepts of White box testing
  - Logic Coverage
  - Control Flow Graph
  - Basis Path Testing
  - Loop Testing
  - Data Flow Testing